feature and for failing to provide an appropriate motivation to combine the teachings of the references. Applicants respectfully request reconsideration.

As Applicants pointed out in response to the previous Office Action, and as again admitted by the Examiner, Piazza does not disclose that his system is applicable to C/C++ program language. To the best of Applicants' knowledge, they are the first to apply compiling techniques to "C/C++ programs designed for a multitasking system into a new C/C++ program designed for a high performance run-to-completion system." See, specification p. 10, lines 9-14.

To repair the admitted deficiencies of Piazza, the Examiner relies on the following statement in McGuire:

> the system and method for variables representing Quantities is implemented in an object oriented language from Curl Corporation. The language allows the creation of objects and has similar capabilities to those found in languages such as Scheme, Lisp and C++." Col. 8, lines 32-27.

The Examiner then reasons, "it would have been obvious to incorporate the teaching of McGuire into the teaching of Piazza to use the object oriented language such as Scheme, Lisp and C++ ... because one of ordinary skill in the art would have been motivated to provide more choices to use when choosing an object oriented language like Scheme, Lisp

and C++ as a development tools to develop the computer system." See, Office Action, p.3.

Applicants respectfully submit that the asserted combination of Piazza and McGuire is improper for failing to provide an appropriate reason to combine the references. Applicants respectfully submit that the Examiner is reading inappropriate meaning into the above quoted statement in McGuire. McGuire's passage merely states that an object oriented language from Curl Corporation has similar capabilities to those found in languages such as Scheme, Lisp and C++. Applicants respectfully submit that McGuire is not saying that Scheme, Lisp and C++ are interchangeable, as the Examiner is apparently reasoning, but rather, that the Curl Corporation language has some similarities with each of those languages. Just because the Curl object language has some unspecified similarities with other languages does not necessarily mean that the other languages are also similar enough to be obvious substitutions for one another.

Other publications in the field support Applicants' position. For example, the attached (Appendix B) pages from a university website[1] highlights the fact that C++ and

[1] Website titled "An introduction to Scheme for C

Scheme are not interchangeable languages. As stated on this website:

> scheme and C each encourage a very different style of programming (and of thinking), and there are a large number of differences between the two languages, … knowledge of C will not only not help you when you learn scheme, it may actually make the process more difficult … a lot of the programs that you have to write [for this course using scheme]… will either not be (directly) writable in C or else would be much more difficult to write in C. (Emphasis added).

Therefore, Applicants respectfully submit that, contrary to the Examiner's suggestion, one of ordinary skill in the art would not be motivated to substitute a compiler for a Scheme/Lisp language for a C/C++ compiler due to the numerous, difficult and conceptual differences between the languages.

In addition, Applicants' respectfully submit that the proposed combination of Piazza and McGuire also fails to disclose or suggest each claimed feature. As discussed above, the Examiner admits that Piazza fails to disclose a compiler for C/C++ programs. McGuire also fails to disclose or suggest a C/C++ program. In fact, McGuire does

---

programmers" accessed by Applicants' undersigned representative on Dec. 2, 2004, at the address: www.cs.caltech.edu/courses/cs1/resources/scheme-for-c-programmers.html.

not even use C/C++ for his claimed invention. All McGuire does is mention that the language he does use, the Curl Corporation object language, has some similarities with C, Scheme and Lisp. Thus, the proposed combination of Piazza and McGuire fails to disclose or suggest at least "a first C/C++ program" as claimed by Applicants. For at least these reasons, Applicants respectfully request that the rejections of claims 1-13 be withdrawn.

I. CONCLUSION

In view of the foregoing, it is respectfully submitted that the present application is in condition for allowance, and an early indication of the same is courteously solicited. The Examiner is respectfully requested to contact the undersigned by telephone at the below listed telephone number, in order to expedite resolution of any issues and to expedite passage of the present application to issue, if any comments, questions, or suggestions arise in connection with the present application.

To the extent necessary, a petition for an extension of time under 37 CFR § 1.136 is hereby made.

Please charge any shortage in fees due in connection with the filing of this paper, including extension of time

fees, to Deposit Account No. 50-0206, and please credit any excess fees to the same deposit account.

Respectfully submitted,

Hunton & Williams LLP

By: _____
Christopher Cuneo
Registration No. 42,450

for
Thomas E. Anderson
Registration No. 37,063

CJC:dms

Hunton & Williams LLP
1900 K Street, N.W.
Washington, D.C. 20006-1109
Telephone: (202) 955-1500
Facsimile: (202) 778-2201

Date: June 28, 2004

APPENDIX A

1. (Previously Presented) A method for transforming a C/C++ program having a first multi-tasking property to a C/C++ program having a second multi-tasking property, the method comprising:

transforming a first C/C++ program having a first multi-tasking property into a data structure;

transforming the data structure to include an explicit multi-tasking transfer of control command;

optimizing the data structure to reduce an amount of program state that is saved at a transfer of control; and

generating a second C/C++ program having a second multi-tasking property using the optimized data structure

2. (Original) The method of claim 1, wherein the data structure further comprises a syntax tree.

3. (Original) The method of claim 2, wherein the step of transforming the data structure to include an explicit

multi-tasking transfer of control command further comprises:

converting the syntax tree to a continuation-passing style (CPS).

4. (Original) The method of claim 1, wherein the first multi-tasking property comprises a property relating to a preemptive multitasking model and the second multi-tasking property comprises a property relating to a run-to-completion model.

5. (Original) The method of claim 1, wherein the first program having a first multi-tasking property operates using a first program language and the second program having a second multi-tasking property also operates using the first program language.

6. (Previously Presented) A system for transforming a C/C++ program having a first multi-tasking property to a C/C++ program having a second multi-tasking property, the system comprising:

a data structure transformer for transforming a first C/C++ program having a first multi-tasking property into a data structure;

a multi-tasking transformer for transforming the data structure to include an explicit multi-tasking transfer of control command;

a program state optimizer for optimizing the data structure to reduce an amount of program state that is saved at a transfer of control; and

a program generator for generating a second C/C++ program having a second multi-tasking property using the optimized data structure.

7. (Original) The system of claim 6, wherein the data structure further comprises a syntax tree.

8 (Original). The system of claim 7, wherein the multi-tasking transformer further comprises:

a converter for converting the syntax tree to a continuation-passing style (CPS).

9. (Original) The system of claim 6, wherein the first multi-tasking property comprises a property relating to a preemptive multitasking model and the second multi-tasking property comprises a property relating to a run-to-completion model.

10 (Original). The system of claim 6, wherein the first program having a first multi-tasking property operates using a first program language and the second program having a second multi-tasking property also operates using the first program language.

11 (Previously Presented). An article of manufacture for transforming a C/C++ program having a first multi-tasking property to a C/C++ program having a second multi-tasking property, the article of manufacture comprising:

at least one processor readable carrier; and

instructions carried on the at least one carrier;

wherein the instructions are configured to be readable from the at least one carrier by at least one processor and thereby cause the at least one processor to operate so as to:

transform a first C/C++ program having a first multi-tasking property into a data structure;

transform the data structure to include an explicit multi-tasking transfer of control command;

optimize the data structure to reduce an amount of program state that is saved at a transfer of control; and

generate a second C/C++ program having a second multi-tasking property using the optimized data structure.

12. (Previously presented) A processor readable medium for providing instructions to at least one processor for directing the at least one processor to:

transform a first C/C++ program having a first multi-tasking property into a data structure;

transform the data structure to include an explicit multi-tasking transfer of control command;

optimize the data structure to reduce an amount of program state that is saved at a transfer of control; and

generate a second C/C++ program having a second multi-tasking property using the optimized data structure.

13. (Previously Presented) A signal embodied in a carrier wave and representing sequences of instructions which, when

executed by at least one processor, cause the at least one processor to transform a program having a first multi-tasking property to a program having a second multi-tasking property by performing the steps of:

transforming a first C/C++ program having a first multi-tasking property into a data structure;

transforming the data structure to include an explicit multi-tasking transfer of control command;

optimizing the data structure to reduce an amount of program state that is saved at a transfer of control; and

generating a second C/C++ program having a second multi-tasking property using the optimized data structure.

An introduction to scheme for C programmers

You must unlearn what you have learned.

-- Yoda

Purpose

This page is an introduction to the scheme programming language for people who have already learned the C programming language, and specifically for students taking the CS 1 class at Caltech (Introduction to Computation). The reason for this page is that this course uses scheme, whereas many students in the class taking the course (most of whom are freshmen) have only programmed in C (and/or related languages like C++ or java) before. This is an issue because scheme and C each encourage a very different style of programming (and of thinking), and there are a large number of differences between the two languages, ranging from trivial syntactic differences to very deep semantic differences. In fact, it is probably safe to say that knowledge of C will not only not help you when learning scheme, it may actually make the process more difficult. Therefore, this page summarizes the major differences between the two languages to make the transition easier. We will assume that you, the reader, are reading this because you are taking CS 1 at Caltech, and we will also assume that you have a reasonably thorough knowledge of C; if not, you have nothing to worry about ;-)

Before we begin, it's important that you realize that we are not just teaching you a new language and a new way of thinking to torture you. Programming languages change and evolve at an astonishing rate; many of the most popular languages around now didn't even exist a few years ago. Programming paradigms change more slowly, but there are already many of these (imperative, functional, object-oriented, logic programming, constraint-based programming, stream-based programming) with more on the way (quantum computation, DNA computation, etc.). If all we taught you was how to program in one language we would be doing you a disservice. What is needed is for us to train you to think in a different and more flexible way so that learning new languages and new paradigms isn't difficult any more. That's what this course is about. It will probably be painful at first, but if you stick with it it will be rewarding.

One thing we will not to do on this page is to indulge in pointless discussion over whether scheme or C is a "better" language; both languages have their place. Scheme does have major advantages as a language for teaching the fundamental ideas of programming, which is why we use it in CS 1. A lot of the programs that you have to write for CS 1 will either not be (directly) writable in C or else would be much more difficult to write in C.

Another thing we won't do on this page is to try and explain the underlying semantics of scheme; we'll do that in class. Instead, this is a practical summary of the differences between C and scheme. You should always be aware, though, that when we say "feature X in C is equivalent to feature Y in scheme", this doesn't mean that they are semantically equivalent; it just means that they are used to achieve the same end.

Finally, if you don't understand something on this page, particularly regarding scheme, don't panic; we've tried to be fairly comprehensive here and many of the aspects of scheme we discuss below aren't covered in class until after the midterm. As always, you should ask your TA, your recitation instructor, or the course instructor if you have any questions.

Menu

The differences between scheme and C can be divided into these categories:

- syntax
- data types
- type systems
- built-in functions
- language features that look different in scheme when compared to C
- language features that exist in scheme but not in C
- language features that exist in C but not in scheme
- things you don't have to worry about in scheme that you do have to worry about in C
- things you don't have to worry about in C that you do have to worry about in scheme
- programming style
- programming environment
- standards and dialects

Also, see the

- links

for more information.

We will discuss each of these individually. Most of these comments also apply to C++ and java as compared to scheme. The main exception is that java, like scheme but unlike C and C++, doesn't have pointers and does have garbage collection. Java and C++ also have a number of other features (exception handling, support for object-oriented programming) that are beyond the scope of this discussion.

Syntax

Syntactic sugar causes cancer of the semicolon.

-- Alan Perlis

Atoms and lists

Scheme has an incredibly minimalistic syntax compared to C. There are basically only two fundamental syntactic forms: individual data objects (often called "atoms") and lists. Atoms consist mainly of numbers and identifiers. Lists are lists of atoms (or other lists) surrounded by parentheses. Some examples:

atoms	lists
1	(x y z)
2.3	(1 2 3 4 5)
x	((1 2) (3 4) 5)

Usually the only other syntactic form you will see in scheme are comments, which start with a semicolon ";" and go to the end of the line:

C	scheme
/* This is a comment in C. */	; This is a comment in scheme.

The semicolon has no other function in scheme. In particular, it does not serve as an end-of-statement marker. There is no multi-line comment operator in scheme either; each comment line must have its own semicolon. It is also quite typical in scheme to have a comment that starts with more than one semicolon; this is just to make the comment look nicer and has no other meaning:

scheme
; This is a comment in scheme.

```
;; This is another comment in scheme.
```

Identifiers

In C, identifiers (function and variable names) can only be made from the letters 'A' to 'Z', 'a' to 'z', the numbers '0' to '9' and the underscore '_'. Scheme is much more liberal about identifiers; in addition to the C characters, you can also use these characters in identifiers:

```
! $ % & * + - . / : < = > ? @ ^ ~
```

The most common of these characters to be used in identifiers are "?", "!", and "-". "?" is typically used in the names of functions that return a boolean (true/false) value (*e.g.* "even?"), "!" is typically used in the names of functions or special forms that change the values of variables (*e.g.* "set!") and hyphens are typically used in function names where C would use underscores (*e.g.* "calculate-result"). These are just conventions and are not enforced by the language. Also, note that the usual arithmetic operators (`+`, `-`, `*`, `/`) are just ordinary identifiers in scheme. We'll talk more about them below.

Also, in scheme, identifiers are **not** case-sensitive, so "foo" and "FOO" mean the same thing when used as *e.g.* a variable or function name. In C, identifiers **are** case-sensitive, so "foo" and "FOO" would represent two completely different names.

Prefix, not infix

Probably the most disorienting aspect of scheme syntax for C programmers is that mathematical operations are expressed in prefix syntax rather than in infix syntax. "Infix" means that the operator (*e.g.* `+`, `-`, `*`, `/`) comes between the operands (*e.g.* numbers or variables) whereas "prefix" means that the operator comes before the operands. Here are some examples:

C	scheme
a + b	(+ a b)
(a - b) * 2	(* (- a b) 2)

This is quite unpleasant for most people at first, but it does have several advantages:

- There is no syntactic difference between an operator expression and a function call with two arguments. In fact, "operators" are just ordinary functions in scheme.
- There are no operator precedence levels (as opposed to the 15 precedence levels in C). For instance, compare these examples:

C	scheme
a * b - c	(- (* a b) c)
a * (b - c)	(* a (- b c))

In the C case, does "a * b - c" mean "(a * b) - c" or "a * (b - c)"? In fact, it means the former, but we need to know that the precedence of "*" is higher than that of "-" to figure this out. In the case of scheme, the two possibilities are syntactically different so there is never any ambiguity. The price we pay for this is that parentheses are never optional in scheme; every parenthesis has to be there, and additional parentheses change the meaning of an expression. For instance, "((foo x y))" is NOT the same as "(foo x y)"); the former expression means to take the result of evaluating the expression "(foo x y)" (which should be a function) and evaluate it again, while the latter expression means to just evaluate "(foo x y)". If this seems confusing, take our word for it; it'll become clearer as we progress in the course.

- In C, operators must have either one operand (unary) or two (binary). (Actually, there is one ternary (three-operand) operator in C, the ?: operator, but we'll ignore it for now). In scheme, many "operators", including the arithmetic operators (which are just function calls, as we mentioned above) can have any number of operands:

C	scheme
a + b + c + d + e	(+ a b c d e)

This is often convenient.

One consequence of prefix syntax is that successive atoms MUST be separated by whitespace (space character(s), tab(s), or newline(s)), or else it would be impossible to distinguish the separate atoms:

C	scheme
a+b+c+d+e /* OK */	(+ abcde) ; no good

In the example above, there is no way to know that "abcde" refers to five different identifiers instead of one identifier called "abcde". You have to put the spaces between them (actually, this is good style in C as well). Similarly, in scheme "a-b" refers to a single identifier and not to "a - b", because the minus sign can be included in identifiers.

Even if you absolutely hate prefix syntax at first, you will probably find that after a couple of weeks it won't bother you any more. Eventually you may even prefer it; it's simpler and less ambiguous.

Functions and special forms

By default, a parenthesized expression in scheme is a function call. The rule for function calls is to evaluate all the arguments and then apply the function to the arguments. C uses the same rule (although with different syntax). There are a small number of "special forms" in scheme that look like function calls but have different evaluation rules (for instance, an "if" expression only evaluates one of two subexpressions). Many of these special forms correspond to similar expressions in C; see below for an item-by-item description.

No special syntax

One confusing aspect of scheme syntax is the fact that there is no special syntax for many of the language features that require special syntax in C. For instance, there is no special syntax for:

- conditionals (if, else)
- function definition
- function calls
- accessing elements of an array

Instead, these language features use special identifiers inside lists. For instance, a conditional expression is a list whose first element is the word "if". See below for more on this.

Data types

Numbers

Scheme has fewer numerical data types than C does. C has a variety of integer data types (short, int, long, unsigned short, etc.) whereas scheme has only one. The scheme integer type is quite powerful, though, in that it can hold arbitrarily large numbers (there is no integer overflow). Similarly, C has two floating-point data types (float and double) whereas scheme only has one. Most schemes (including DrScheme) also have built-in support for rational numbers and complex numbers, but we won't be using them in CS 1.

Booleans

Unlike C, scheme has a real boolean type. "True" is designated by "#t" and "false" by "#f". DrScheme also lets you use "true" and "false" in place of "#t" and "#f" on many of the language levels, but don't do it: it isn't portable to other versions of scheme.

Like C, scheme is not too picky about true values; any value that is not a false value is considered to be true if it's used as a boolean.

Characters

Both C and scheme have a character data type:

C	scheme
'a' 'b' 'c' '\n' ' '	#\a #\b #\c #\newline #\space

As you can see, some characters have mnemonic names *e.g.* #\newline.

Strings

In C, strings are just arrays of characters (chars). In scheme, strings are a basic data type. The syntax is the same, except that scheme doesn't necessarily support the C escape sequences like "\n". Many schemes (including DrScheme) do support this syntax, though.

Symbols

Scheme also has a data type for symbols. See below for more on this.

Lists

Lists are a built-in data type in scheme but not in C. These lists are singly-linked lists, which means that from a given element you can only go in one direction along the list. Other kinds of lists (*e.g.* doubly-linked lists) are not built in to either C or scheme, but can easily be defined in either language. You will learn all about how to create and manipulate lists in CS 1 or in any scheme textbook, so we won't go over that here.

Arrays

Both C and scheme have arrays, but in scheme, arrays are called "vectors" and can contain objects of any type, whereas in C they can usually only contain objects of a single type. See below for more about arrays.

Type systems

C is a *statically typed* language. That means that all variables have to have a pre-declared type, and it's an error to assign a value of one type to a variable that has been declared to have a different type (or at least, it generates a compiler warning). In addition, all variables must be declared before use. In contrast, scheme is a *dynamically typed* language. In scheme, *values* (data objects) have types, not variables. A variable can store values of any type, and can store a value of one type at one time and another type at another time. Some people say that scheme is "weakly typed" because of this, but that's not true; it's simply that type information is used differently. Variables in scheme do need to be declared before use, but their type doesn't have to be specified. Example:

C	scheme

`int x = 0;`
`x = 1;`
`x = "foo"; /* compiler complains */`
	`(define x 0) ; no type specified`
	`(set! x 1)`
	`(set! x "foo") ; OK`

Note that many type errors in C don't result in errors but instead in compiler warnings (as in the above example). Thus, C's type checking is quite loose. We might say that C is only weakly strongly-typed ;-)

Built-in functions

Scheme has a fairly large number of built-in functions that can be used directly in programs (*i.e.* without the equivalent of a "#include" statement in C). Examples include basic math functions like `sqrt` (square root), `sin` (sine), `cos` (cosine) etc. These functions are NOT KEYWORDS; they are just ordinary functions. See the documentation in the links below for more information on these functions. Example:

C	scheme
`#include <math.h>` `sqrt(10.0);`	(sqrt 10.0)

Language features that look different in scheme when compared to C

Many language features are present in both C and scheme but look quite different. We will show examples of each of these features without much discussion. For more details on the scheme code see the links at the bottom of this page.

- global variable definitions:

  These occur at the top level of the program (outside of function bodies).

C	scheme
`int i = 10;`	`(define i 10)`

- local variable definitions:

C	scheme

`{`
`  int i = 10;`
`  int j = 20;`
`  /* more code */`
`}`
	`(let ((i 10)`
	`  (j 20))`
	`  ; more code`
	`)`

`{`
`  int i = 10;`
`  int j = i;`
`  /* more code */`
`}`
	`(let* ((i 10)`
	`  (j i))`
	`  ; more code`
	`)`

Note that if you have multiple local variable definitions where subsequent definitions depend on previous ones, you have to use "let*" instead of "let" in scheme. In C the syntax is identical in both cases.

- assignment statements:

C	scheme
`i = 20;`	`(set! i 20)`

Note that the normal style of scheme programming (see below) makes most assignment statements unnecessary. Assignment statements are thus much rarer in well-written scheme code compared to C code. In fact, for the first half of CS 1 you should pretend that assignment statements don't even exist.

Another thing to note is that the return value of an assignment expression in scheme is unspecified, so you can't do the equivalent of "a = b = c = 0;" in C:

C	scheme

`a = b = c = 0;`	`(set! a 0)`
	`(set! b 0)`
	`(set! c 0)`

- sequential expressions:

C	scheme

`printf("hello, world\n");`
`j = 10;`
	`(begin`
	`  (display "hello, world")`
	`  (newline)`
	`  (set! j 10))`

The return value of the "begin" statement is the value of the last statement evaluated. In this case, the last statement is a "set!" (which has an "unspecified return type" which means you shouldn't count on anything being returned), so you shouldn't use the return value for anything. In C, blocks of statements don't have a value, so the problem doesn't come up.

- function calls:

C	scheme
`foo(a, b, c)`	`(foo a b c)`

- function definitions:

C	scheme

`int foo(int x)`
`{`
`  return x * 2;`
`}`
	`(define (foo x)`
	`  (* x 2))`

`int foo(int x)`
`{`
`  int y = 10;`
`  return x * y;`
`}`
	`(define (foo x)`
	`  (let ((y 10))`
	`  (* x y)))`

`int foo(int x)`
`{`
`  printf("x = %d\n", x);`
`  return x * 2;`
`}`
	`(define (foo x)`
	`  (display "x = ")`
	`  (display x)`
	`  (newline)`
	`  (* x 2))`

Note that scheme doesn't have a "return" statement. The last expression evaluated in the function body is the value of the function.

- conditional expressions:

C	scheme

`if (a > 0)`
`{`
`  printf("Woo hoo!\n");`
`  a = 1;`
`}`
`else`
`{`
`  printf("Aw shucks!\n");`
`  a = -1;`
`}`
	`(if (> a 0)`
	`  (begin`
	`  (display "Woo hoo!")`
	`  (newline)`
	`  (set! a 1))`
	`  (begin`
	`  (display "Aw shucks!")`
	`  (newline)`
	`  (set! a -1)))`

`(a > 0) ? 1 : -1`	`(if (> a 0) 1 -1)`

`if (a > 0)`
`{`
`  printf("greater than 0\n");`
`}`
`else if (a < 0)`
`{`
`  printf("less than 0\n");`
`}`
`else`
`{`
`  printf("equal to zero\n");`
`}`
	`(cond ((> a 0)`
	`  (display "greater than 0")`
	`  (newline))`
	`  ((< a 0)`
	`  (display "less than 0")`
	`  (newline))`
	`  (else`
	`  (display "equal to zero")`
	`  (newline)))`

Note that after each of the tests in a "cond" statement there is an implicit "begin" *i.e.* you can put several statements before the closing parenthesis and they will all be executed sequentially. If you want to do this with an "if" statement you have to put the "begin" in explicitly.

- iteration constructs: do

C	scheme

`for (i = 0; i < 10; i++)`
`{`
`  printf("i = %d\n", i);`
`}`
	`(do ((i 0 (+ i 1)))`
	`  ((= i 10) #f) ; return #f at the end`
	`  (display "i = ")`
	`  (display i)`
	`  (newline))`

Explicit iteration statements using "do" are extremely rare in well-written scheme code; instead, the normal style is to use recursion for iteration (see below).

- operators:

  Some operators are different in scheme and C.

C	scheme
`==`	`=`
`!`	`not`
`a != b`	`(not (= a b))`
`a++;`	`(set! a (+ a 1))`
`a += 10;`	`(set! a (+ a 10))`

  (and similarly for "--", "-=", "*=" etc.).

- arrays:

  Arrays are called "vectors" in scheme.

C	scheme
`int i[] = { 1, 2, 3, 4, 5 };`	`(define i (vector 1 2 3 4 5))` `(define i #(1 2 3 4 5)) ; equivalent`
`i[2]`	`(vector-ref i 2)`
`i[2] = 10;`	`(vector-set! i 2 10)`

  The scheme syntax for vectors is admittedly quite verbose.

Language features that exist in scheme but not in C

Scheme includes several language features that are not found in C:

- functions are data

  Probably the most profoundly different feature of scheme relative to C is that functions are "first-class" data in scheme but not in C. What "first-class" means is that in scheme, functions are data just like numbers or strings. Functions can be passed as arguments to other functions, they can be returned as the result of a function, and they can be created inside a function. Here is an example that

shows this:

C	scheme
No equivalent!	`(define (adder n)` `  (lambda (x) (+ x n)))`

What the "adder" function does is to create a new function of one argument that adds "n" to the argument. It's used like this:

```
(define add2 (adder 2))
(add2 5) ; result: 7
```

This ability to create functions on-the-fly is very powerful, and we will be exploring this in detail in CS 1. The mysterious word "lambda" basically just means "make a function", so

```
(lambda (x) (+ x n))
```

means "make a function of one argument called x which adds x to n and returns the result". It turns out that all function definitions in scheme are actually lambda expressions, so that

```
(define (f x) (* x 2))
```

is just a syntactic shorthand for

```
(define f (lambda (x) (* x 2)))
```

The word "lambda" comes from the lambda calculus, which is an abstract model of computation which furnishes the theoretical foundations for the scheme programming language. But that's another story. You should think of a lambda expression as a function without a name; the "define" expression gives it a name (it binds the name to the function). C does not have anonymous functions.

Functions which accept functions as arguments are known as higher-order functions. They have many uses; here is a trivial one:

```
(map  (lambda (x) (* x 2))  (list 1 2 3 4 5))  ; result: (2 4 6 8 10)
```

Here, "map" takes a function as its argument and "maps" the function over each element of a list. The result is a list where each element is twice as big as the corresponding element of the input list. You'll see many more examples of higher-order functions and their uses in the course (and in SICP, the course text).

It should be noted that C contains a very primitive version of this capability: function pointers. Function pointers allow a C programmer to pass a function as an argument to another function. However, it isn't possible to define a function inside the body of another function in C (at least not in standard C; `gcc` does allow this), so this limits the usefulness of function pointers considerably.

- functions can be defined inside functions

  There are a variety of ways of defining functions inside functions in scheme. The simplest way is to use an internal "define":

```
(define (foo x)
  (define (bar y) (* y 2))
  (* (bar x) 3))
```

  This is a totally trivial example, but nested function definitions can be quite useful, as you'll see in the course.

- lists are a built-in data type

  As mentioned above, lists (more specifically singly-linked lists) are a fundamental data type in scheme but not in C.

- symbols and literal lists

  Scheme has a special data type for symbols and symbolic expressions. They are created using the "quote" special form:

```
(define a (quote b))
(define a 'b) ; the same thing
```

  The single-quote character is syntactic sugar for the "quote" expression, so "'b" is the same as "(quote b)". But what is "(quote b)", anyway? It's just the symbol "b", unevaluated. Basically, what "quote" does is to switch off the scheme evaluator. If there was no "quote", then scheme would try to evaluate "b" in the current environment. We call "(quote b)" a symbol (in this case, the symbol "b"). You can also quote whole expressions e.g.

```
(quote (1 2 3 4 5))
'(1 2 3 4 5)  ; the same thing
(list 1 2 3 4 5) ; equivalent
```

  In this case, "'(1 2 3 4 5)" is just the literal list "(1 2 3 4 5)", which is also the result of evaluating the expression "(list 1 2 3 4 5)". This is a convenient shorthand for creating literal lists. There are lots of other uses for quoted expressions, but they're beyond the scope of the course.

- continuations

  Continuations are pretty advanced, and you don't have to know about them for CS 1. On the positive side, continuations can be used to implement arbitrarily complex control structures. On the negative side, too much programming with continuations can make your head explode ;-) Scheme supports continuations as first-class objects. If you're curious, see an advanced scheme textbook or do an internet search.

Language features that exist in C but not in scheme

The following constructs in C have no direct counterparts in scheme:

- pointers

  Scheme does not permit direct manipulation of pointers (variables that hold memory addresses) or pointer arithmetic the way that C does. However, in some sense every value in scheme is a pointer to an object, so this doesn't cause problems in practice (unless you need direct access to the machine's memory for some reason, in which case scheme is not the right language for the job). Pointers are a notorious source of bugs in C; not having them in scheme makes a programmer's life much more pleasant.

- structs

  Standard scheme does not have a struct-like construct. However, virtually all scheme dialects do have such a construct (for instance, DrScheme does). It is quite easy to roll your own struct-like construct in scheme with a little effort.

- "switch" statements

  You can achieve the same effect as a switch statement in C using a cond statement:

C	scheme

`switch (a)`
`{`
`  case 0:`
`  printf("zero\n");`
`  break;`
 	
`  case 1:`
`  printf("one\n");`
`  break;`
 	
`  default:`
`  printf("whatever\n");`
`  break;`
`}`
	`(cond ((= a 0)`
	`  (display "zero")`
	`  (newline))`
	`  ((= a 1)`
	`  (display "one")`
	`  (newline))`
	`  (else`
	`  (display "whatever")`
	`  (newline)))`

- bitwise operators

There are no bitwise operators in standard scheme. C is a better language than scheme for low-level bit manipulation.

- unsigned integers

  There are no unsigned integer types in scheme.

Things you don't have to worry about in scheme that you do have to worry about in C

Generally speaking, scheme is a much safer language to program in than C. Large numbers of bugs that typically occur in C programs either can't happen at all in scheme, or else they can't happen without giving rise to an immediate error which the scheme environment reports to the programmer. Examples include:

- memory leaks

  Memory allocated in a scheme program does not have to be explicitly freed, because the scheme run-time system does that for you. This is called garbage collection, and it makes programming much less of a hassle than in C, where you have to explicitly free all memory that you allocate, or else you are likely to run out of memory.

- memory corruption and array bounds violations

  In C, if you try to access the 100th element of a 10-element array, anything can happen. You might get a core dump (*i.e.* your program/OS may crash), or your program may continue and then mysteriously fail at a later time. In scheme, an error is reported immediately.

Things you don't have to worry about in C that you do have to worry about in scheme

There really is only one thing you have to worry about in scheme that you don't have to worry about in C: type checking. In C, the compiler checks that all operations are performed on values of the correct type and complains if they're not (more or less; it's possible to disable this using type casts). In scheme, type checks are done at run time, so that you can write a function with incorrect type usages, and no error will be reported until you run that function. Example:

C	scheme

```
void bad()
{
    char s[] = "foo";
    int result;

    result = 1 + s;  /* generates a compiler warning */
}
```

```
;; This is perfectly legal scheme:
(define (bad)
  (+ 1 "foo"))

;; We get an error when we call the function:
(bad)
```

Note, however, that even flagrant type errors like trying to add a string to an integer only result in compiler warnings, not outright errors. The type checking in C is actually quite loose (the same code would give a compiler error in C++), so the type checking in C doesn't help as much as you would think.

Programming style

> *You have taken your first step into a larger world.*
>
> -- Obi-wan Kenobe

Scheme encourages a very different style of programming than C does, and this is a major obstacle for beginning scheme programmers who are accomplished C programmers. Suffice it to say that if you think that all programming languages are basically the same, you are in for a very unpleasant shock. The style of programming that scheme encourages is usually known as the functional programming style. This has a number of features:

- Assignment statements ("set!" in scheme) are used as little as possible.
- Explicit iteration statements ("do" in scheme) are used as little as possible; instead recursion is used to implement iteration. A recursive function is one that calls itself.
- Functions are written so that they do not modify any global variables or the arguments that were passed in to them. They only communicate through the outside world by the value they return. In other words, they behave like strict input-output devices, like mathematical functions. This property is called referential transparency; the function does the same thing no matter what the circumstances are when it is called.
- Mutable data structures like arrays (vectors) are used as little as possible. Instead, persistent data structures (often lists) are used.

Persistent means that you act on *e.g.* a list by generating a new list, not by modifying the existing list. This is a consequence of not modifying the arguments to functions, and it prevents a large number of bugs. However, operations on persistent data structures are sometimes less efficient than operations on mutable data structures for certain algorithms, so the benefits must be weighed against the costs.

- Higher-order functions are used extensively.

In contrast, the style of programming that C encourages (forces?) is called the imperative programming style, and involves a lot of mutable state (variables, arrays, structs, etc.). Experienced C programmers who have not programmed in a functional style usually can't believe that it's even possible to program in any other way than the imperative style. As you will see in the course, it is indeed possible, but it takes some getting used to. The usual reaction is then something like "OK, I see how to do this, but why would you want to program that way? Isn't it making simple things needlessly complicated?" It turns out that programming in a functional style is a powerful way to write programs that are correct by construction. Programs with a lot of assignment statements and explicit iteration statements are much more vulnerable to various kinds of bugs (*e.g.* array indices that are off by one) than programs written in a functional style. However, this style of programming is bound to seem unnatural at first. Bear with it; there is a point to it.

Having said that, it's important to realize that it is possible to write code in scheme in a completely imperative style as well. One of the big advantages of scheme is that you can use multiple different programming styles (whichever one suits the problem best). In contrast, it is very difficult to program in C in anything other than a purely imperative style.

Programming environment

Programs written in C have to go through a somewhat elaborate compiling and linking cycle before they can generate a program that can be run. In contrast, most scheme implementations provide an interactive programming environment where code can be typed in and executed immediately (DrScheme is an excellent example of this). This is very convenient when developing code because you get feedback much more quickly than you do when programming in C. Some scheme implementations also include compilers that can produce stand-alone executables. If we have time, we may use one of these compilers at some point in the course.

Standards and dialects

C is pretty much a fully standardized language. There are minor differences between pre-ANSI C and ANSI C, and most compilers support a few extra features that aren't standard (*e.g.* inline functions, "//" comments; the GNU C compiler `gcc` supports a large number of other extensions as well), but basically, C is the same language everywhere. The situation is different with scheme. There is a fundamental core of

functionality that a scheme implementation has to support in order for it to call itself scheme; this is the scheme standard. The current standard is called "R5RS"; it's the fifth revision of the scheme standard. However, there are a lot of fundamental features that are not covered by the standard. They include things like scheme's equivalent to C's structs, module systems, exception handling systems, and object-oriented extensions to scheme. Because scheme is so flexible, many such extensions exist. Unfortunately, almost all of them are incompatible with one another. This is not a problem for our purposes because we will only be needing the core R5RS scheme for the course (and not even all of that). However, you should be aware that each particular implementation of scheme is a dialect with a common core.

Links

- The Caltech CS 1 home page.
- The current definition of the scheme language.
- The scheme home page.
- The DrScheme home page.
- Structure and Interpretation of Computer Programs, the course text for CS 1 (also known as SICP ("sick-pee") for short). The entire text is now online! SICP is probably the best book ever written about computer programming. It is not an easy read by any standard, but if you get through it it will change the way you think about programming forever.
- How to Design Programs, a textbook (also online!) for an introduction to programming that uses scheme. This is a much gentler introduction to programming than SICP, but it is still very thorough, and it includes some interesting material that SICP doesn't have. Warning: the dialects of scheme used are often highly non-standard, and the book uses a succession of dialects, each adding new concepts to the previous version.